Cluster Reliability Subsystem

Jim Kowalkowski

1 Introduction

The LQCD compute cluster is going to be very large and will need to be available 24 hours a day. The cluster should insure that resources are used to best possible extent and attempt to complete started tasks in the presence of hardware and software failures (be fault resilient). The administration and support team has typically been small. As the cluster grows, the load on this team also grows. Without a set of tools to automate routine administrative and problem solving tasks, the team could spend too much time doing them manually. An automated system will also improve response time in routine problem solving.

The question being addressed in this document is: How do we increase the availability, utilization, and reliability of the computing cluster and also reduce the administrative workload associated with these activities? The purpose of this document is to define a cluster monitoring and control subsystem that can be used to solve this problem.

Examples of things that can affect availability and performance include:

- power outages - scheduled and unscheduled
- job failures due to failing or failed hardware
- scheduling jobs on faulty nodes
- decreased performance due to hardware deterioration
- decreased performance due to external influences (e.g. air quality)
- inability to diagnose problems (e.g. hardware, OS, batch tools)

One job of this subsystem is to recognize that we are in situations such as the ones described above and then react to either correct the problem or manipulate the system so that we can live with the problem. Correcting a problem can mean cleaning up after a failed job, or rebooting a node that is in a bad state, or revalidating a piece of hardware. Living with a problem can mean removing a node from service. A second purpose of this document is to list a basic set monitoring and reaction components that should be available immediately.

1.1 Scope

The goal of this subsystem is not to produce a comprehensive list of all possible problems that can occur on a cluster, but only to produce a list that is a starting point based on what we already know to be problems. This initial list will help us

develop the correct abstraction for later iterations and additions, and also provide the project will a usable product instead of just a toolkit.

The initial focus for direct application of this system will be on hardware-related problems and necessary interactions with core software infrastructure pieces such as the operating system, the batch queuing system, and the job scheduler.

This subsystem will not include development of facilities for which there already exists a reasonable solution to a particular problem. An example is a message passing system.

1.2 Rationale

The introduction has already stated some of the reasons for creating this subsystem.

1.3 Terminology

Software framework: The software that ties all functionality necessary to operate the system together and enforces uniformity in configuration, data exchange, and execution semantics between all the components under its control.

Pluggable software component: a body of executable code that conforms to an API of a software framework and is callable from that framework. This code can be added to or removed from the running instance of that software framework on demand. This code does not need to be part of the core or base installation of the software framework. Each of these components is independent of the others. They communicate using abstractions provides by the software framework.

Monitor: a software component that watches some part of the system being monitored and reports information about that part.

Reactor: a software component that subscribes to information reported by monitoring or other report generating software and takes action based on the contents of one or more reports.

Event: A structured piece of information in the form of a message describing an occurrence within the system e.g. measured quantity or state change.

Sensor: A software component that makes measurements from the environment, formats the information input an “event”, and sends the event into the system (plugin).

Filter: A software component that prevents events from entering a particular part of the system.

Analyzer: a software component that takes in events and attempts to draw conclusions about them (plugin).

Effector: A software component that takes actions based on results from Analyzers or input from Sensors (plugin).

1.4 Basic feature overview

The system must allow users to define software components that monitor any activity on the cluster either periodically or asynchronously using interrupts (e.g. signals, exceptions) and report findings in a standardized way. The system must also allow users to define software components that subscribe to specific report types and can react to the information. A software framework that allows all these user-defined components to be installed and removed dynamically across the entire system from a central authority is highly desirable.

Some basic features of this subsystem include:

- Definition of the types of information that must be communicated and the general format of that information,
- Definition of APIs for communicating information and the creation of monitors and reactors, including programming language bindings that allow creation of monitors and reactors in languages known to the users,
- Definition of a work environment for writing, testing, and releasing new reactors and monitors,
- Definition of a core or basic set of problems that must be addressed, and the monitors and reactors and the data structures that they will communicate in order to solve the problems,
- Recording of monitoring information and actions taken so the information can be used in the future,
- Administration tools that allow for single-point release distribution and installation, and control of the runtime environment,
- A configuration system that allows for uniform parameter setting for reactors and monitors and allows for tuning to adjust the performance impact that this system will have on applications.

We also know of techniques within operating system process schedulers for providing time slices that are synchronized across the entire cluster. The advantage of such an arrangement is that all monitoring activities can be performed at nearly the same time on all nodes, meaning that individual program instances within one job will not be starved for information from other program instances that are waiting for administrative (monitoring) tasks to complete. We would like to know if such an approach can be used to improve the performance of the LQCD cluster and if so, how to incorporate it into this system.

1.5 Organization of this paper

This paper will discuss most of the areas that need to be addressed for this project. Those areas include:

- Development environment (repository structure and tools)
- Build/Release strategy (including tools and management)

- Installation/Deployment
- Unit test strategy
- Setup and use of a test stand
- Configuration management for a running system

Most of the diagrams in this paper follow the following rules. Green ovals are functions of external system that we rely on. Yellow ovals are functions that we own. The grey boxes are collections of files. The arrows are used to show data flow between functions and files (one can also say that they represent a “uses” relationship between two components). Grey ovals represent functions that are plugins, isolated functions are bound to the system at run-time.

1.6 Questions this paper should address

During initial discussions of this system, the following list of things that need to be addressed:

- How does configuration work? (at the system and module level)
- Who talks to whom? (addressing within messages)
- What is the Effector API and what is its purpose?
- What is the Coordinator’s purpose?
- How should sampling work? (push/pull, triggered on changes)
- Should we support dynamic sampling rates based on messages arriving?
- How will testing work? (Sensors as data generators)

This document should be modified to include information about all of these things.

2 A brief survey

Here we are going to briefly address the following questions:

- What does the current system do to address the problem?
- What do other similar (HEP/LQCD) installations do to handle the problem?
- What do other application areas do to address the problem?

We recognize that there are several existing products that will most likely be useful in constructing this subsystem. Examples of products include: Ganglia, MonALISA, Nagios, NetLogger, RRDtool, various SNMP tools, OpenNMS, and Aware.

2.1 The current system

There exists a process (agent) on each worker node called a *Health*. It is a script written in Perl. It is a daemon that periodically runs a long series of code blocks that check various things (about every 10 minutes). These code blocks read temperatures, load, fan speeds, disk information and many other things from the OS. They verify that processes are running, such as PBS. They verify that file sys-

tems are up. Each can take corrective action. They watch for out of control processes, filled file systems, or other similar activities. They also watch if MPD becomes stuck. All collected data and announcements from the worker are sent to the head node using the syslog facility.

The syslogd on lqcd.fnal.gov (the head node) collects messages from the worker and writes them to log files. Another daemon process called *nanny* scans the log files and parses out any interesting information. The nanny also monitors various services on the head node. It sends email message to alert people of recognized conditions and automatically takes some actions such as restarting processes if there are problems. The performance measurement messages are sent to the hplot/MRTG tools for network throughput trend plotting and web page building. A second daemon called *qcdmonitor* watches the log files for temperature measurement data messages. This process has the responsibility of shutting down the entire cluster if a majority of the machines are starting to run too hot.

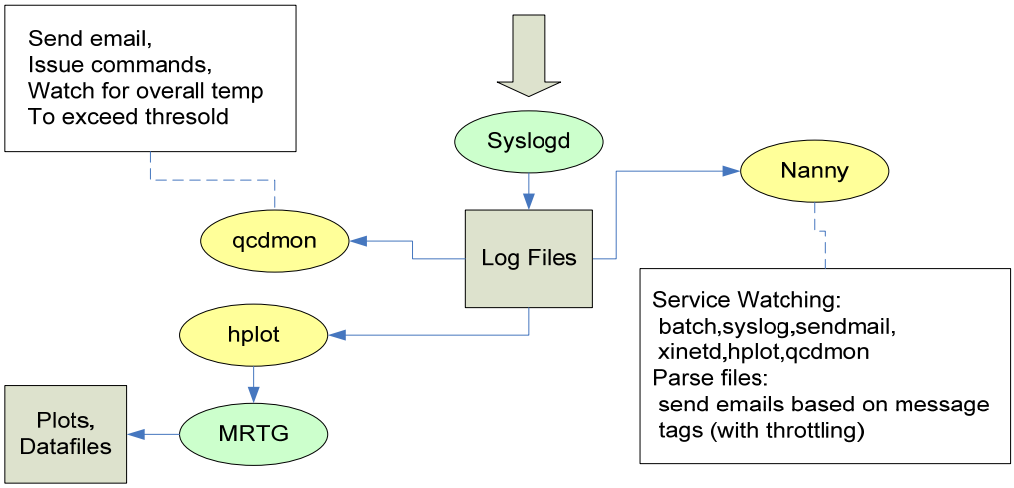


Figure 1

Figure 1 shows the functions of the current system running on the head node and there relationship to one another. The important feature here is that log files appear in a file system directory and are periodically scanned for interesting information by daemons. The syslog facility and level direct messages to particular log files.

Figure 2 shows the functions that the current worker supplies. The diagram depicts them at plugins according to the conventions used in this document, but really they are fixed code blocks with in the script. The IPMI information for this worker is read directly on the worker. The health script watches disk, swap, and

NFS mounts (aspects of storage monitoring), it also watches for PBS to be stuck (service monitor), measures uptimes, makes sure that the IB drivers to be working, and checks that the CPU state is OK (hyperthreading is off and the processor is running at the correct speed). The physical attributes monitor reads fan speeds and temperatures.

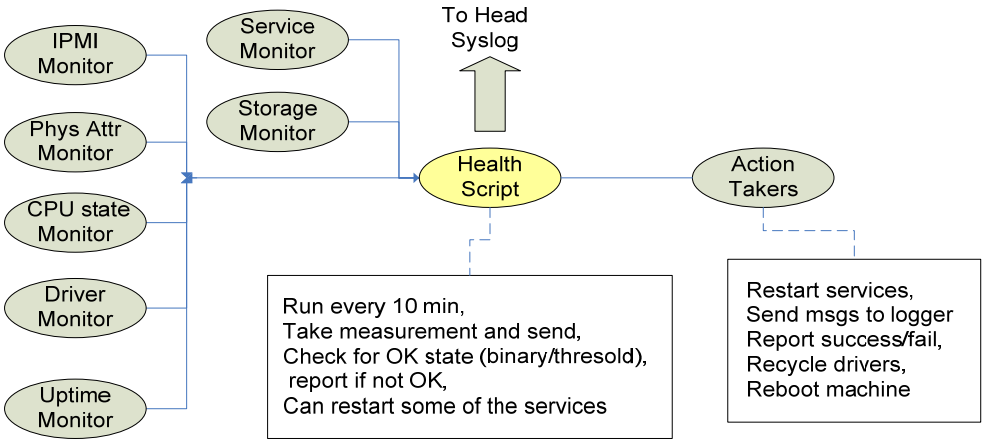


Figure 2

Nodes currently cannot be taken out of service (uninstalled from PBS availability). Hung nodes are a problem; they are only noticed by failure to be able to schedule jobs. We want to automate restarting these failed nodes. Some failures are related to temperature. In the case of hung jobs where node restart is required, entire MPI jobs get hung and need cleanup. Notifications of failures and actions need to be sent out. If the head node is hung (head of MPI job), the job is really stuck because of the relationship between a job and PBS - in other words, PBS is stuck with regards to this job. Restarting jobs based on policy is useful also. Automating the boot sequence is also valuable. There is a verification job that needs to run under power-up or on other occasions.

2.2 Other HEP/LQCD experiment experiences

To be filled out.

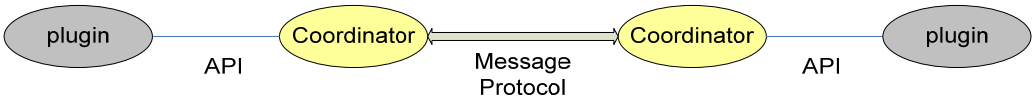
2.3 Solutions in other clusters

To be filled out.

3 Overview

We are considering a system where pluggable components hook into a message distribution system for routing and delivery to other pluggable components. For

lack of a better term, for now we will call this distribution process the **coordination framework**.



From a high-level infrastructure point of view, this core part or fundamental part of the system is composed of three things: the coordination framework, a basic set of plugins to minimally provide existing functionality, and a set of databases. Additional features such as automated diagnostic tools will be built on top of these core pieces.

We will be involved in the defining and writing plugins, which are essentially measurement, diagnostic, and recovery scripts that conform to the coordinator API. Any instance of a plugin is under direct control of the coordinator. The coordinator causes the plugins to come to life and to be called upon to perform their task. The coordinator is responsible for message routing and delivery to final destinations.

The databases for this project are important. The information stored will be used for problem diagnosis and to better understand the operation characteristics of the cluster. They will also be used to know the state of the hardware and software within the cluster. The major database types include: monitoring (data from the messages arriving from the coordinator), state (what hardware is working), inventory (installed software and hardware and its name and address), and bookkeeping (attached directly to coordinators containing addressing information, scripts, versioning, and state). The monitoring data is placed into the monitoring database by a plugin. The same is true for the state changes or status changes caused by arriving messages. The bookkeeping database is used directly by the coordinator to maintain state and configuration information. Deployment and configuration tools populate, distribute, and maintain, the bookkeeping databases.

4 Requirements

Describe how this subsystem will be used by outside actors. Include constraints imposed by outside systems and any other important factors governing its design and implementation.

4.1 Actors

Describe here the set of characters that will be interacting with this system. This list should not include internal actors.

4.2 The Major Inputs and Output

Describe what the system needs to do its work and what it produces.

4.3 Behavioral requirements (use cases)

List the steps necessary to perform each important task associated with this system. Do not include any implementation details here, only brief statements of actions and responses for each one. Each use case should only cover one specific task. Below is a template for use cases.

4.3.1 Prototype use-case

Task	Name of this task
Level	Summary/user goal/sub-function
Goal	Stated as a short active verb phrase
Actor	Who does this
Trigger	Why this is happening
Preconditions	Things which must exist before the use case can start, and any particular state the overall system must be in to allow performance of this case
Post-conditions	New state which exists at successful completion of use case
Description	Steps (numbered) to complete the task. This should include a narrative description of the manageable series of steps that make up the use case
Nonstandard Flow	Exceptions (error conditions) to the standard flow or alternative success routes.
Comments	Link to other information regarding with use case

4.3.2 Create a Sensor Plugin

Task	Create a new Sensor plugin
Level	user goal

Goal	Write a new plugin that functions as a sensor and make it available for use.
Actor	Developer (administrator acting in this role)
Trigger	New measurement is needed
Preconditions	None
Post-conditions	New messages and plugin are available and identified in the inventory database
Description	1. Define the message type and payload format. Register the message. 2. Identify the parameters necessary to configure this plugin. 3. Create a new directory for storage into CVS using the sensor prototype plugin. 4. Write plugin script according to the coordination framework API. Involves writing an initialization routine that takes addressing information and configuration parameters, writing functions to announce required message types (produced and consumed), to produce data, and to access messages. 5. Write a test for the plugin and run it 6. Check everything into CVS and tag it with a version number. 7. Use the administrative tools for the inventory database to declare that the plugin is available for use.
Nonstandard Flow	None
Comments	None

4.4 Constraints

List any additional non-functional requirements here or reference them here. Examples include known external interfaces or protocols or performance constraints.

We are required to use Python as language for writing plugins.

Desire to have sensor and effector processing synchronized across all workers (assuming clock sync with NTP good enough).

5 Architectural Overview

This section can contain a series of diagram illustrating system parts and their relationships.

5.1 Roles

The roles or functional units are defined and described in this section.

5.2 Functional unit or Component block diagram

Figure 3 shows the components we want to see active on the head node in the final system. The grey ovals to the left of the coordinator represent data producing plugins that we want to have. They provide information to the coordinator in the form of messages. The grey ovals to the right of the coordinator represent plugins that consume messages and act upon them. Some of the plugins, in turn, can send new messages back to the coordinator. The functionality shown here is greater than that of the current system.

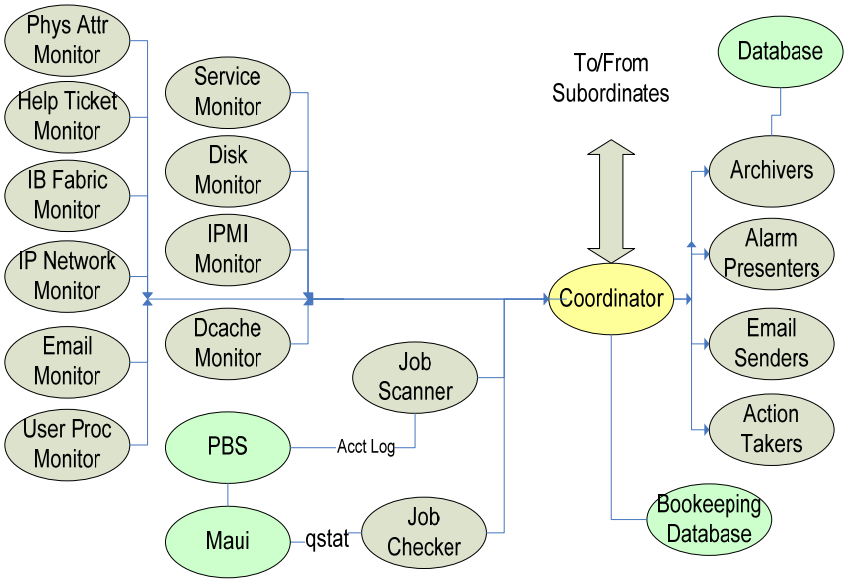


Figure 3

Each monitor should perform a specific task. Some of the monitors shown may actually turn out to be a set of plugin. An example would be the *Service Monitor*; if there difficult types of services that end up being monitored, it might be advantageous to create one monitor per service type to simplify development, testing, and configuration. Because the IPMI data is available on the network within interaction with the processor on a node, we can create an *IPMI Monitor*

independent of the node being monitored. This has the advantage that the CPU on a worker does not need to be involved. The *Dcache monitor* reports performance data and status information from the dcache nodes. Other interesting additions are the *email monitor*, which watches for relevant information arriving on the administration and user email lists, and the hooks into the batch system and scheduler, for producing messages about jobs.

The *action-takers* are actually a set of plugins that may function as both *analyzers* and *effectors* (plugins on the right). It is probably easier to create them as a pair of plugins: one that does the analysis to discover that something is wrong, therefore producing a message about the problem, and one that takes action based on the receipt of a message containing information about an observed condition. There should be one action-taker per unique thing that needs to be taken care of given a set of messages. A good example of an analyzer/action-taker is an *analyzer* component that watches the temperature measuring and concludes that the farm has exceeded a given threshold. The *effector* component watches for the exceeded-threshold message and issues the proper instructions to shut down the cluster. The *archiver* plugins turn incoming messages into data that will be stored in a relational database.

The *bookkeeping* database is generated as part of the configuration/release procedures. It contains information that the coordinator needs to operate, such as the plugins that need to be active and their specific configuration data, identity information (addressing and names), and parent child connection requirements.

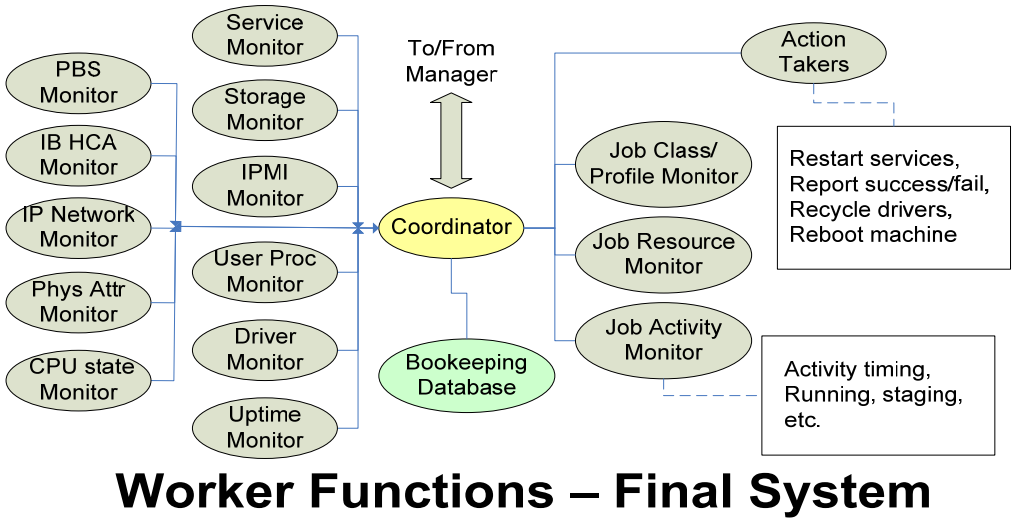


Figure 4

Figure 4 shows the components we want to see active in a worker. The general description of the types of plugins matches that of the head node and will not be

repeated here. An interesting addition here is the job profile/resource monitoring plugins. There components report when different kinds of job activities or phases are occurring and how long they take. Examples are staging data, running, or saving results.

5.3 Physical unit block diagram

Show here the known hardware configuration that is to be built or is available for use.

5.4 Deployment scenario

How do functional units and components map to physical devices?

6 Component Interfaces

Expand the interfaces of the components shown in the previous section. Include relationships to other components here.

7 Protocols

Describe known elements of any protocol involved in data exchanges, external or internal to this subsystem, and the types of messages or data that may be exchanged.

Show important invocation or message exchange timing sequences here. Show what parts of the interfaces are used by other components.

All messages sent need to include the following information.

Len	TimeStamp	Sender	creatorType	msgType	payload

Where:

- Len = length of the message in bytes
- TimeStamp = time that the message was generated (seconds/milliseconds)
- Sender = address of the sender
- CreateType = The type of plugin that created it (sensor, analyzer, effector)
- MsgType = The type of message that this is (phys attr, resource usage, etc.)
- Payload = the data specific to this type of message

Note that a destination is not given in the message. Messages are received based on the type that they are, not on who sent them. The type information may need to be expanded to include a list of more general categories that the message belongs to (e.g. fan speed and CPU temperature are physical attribute measurements). Message types may need to be globally registered to make the problem easier to solve.

8 Discussion

This section captures discussions and information that lead to the current architecture view and component organization.

8.1 Decisions and Choices

Other solutions that were considered and rejected should be briefly summarized here along with arguments for and against them. The purpose of doing this is so old arguments do not continually resurface.

8.2 Rationale

Why the current architecture and component interfaces are appropriate for the problem.

8.3 Implications resulting from Choices

Additional constraints that are imposed on the whole system or this system as a result of choices made here.

8.4 Resulting rules

Include all things that must be true in the system and rules that must be followed while the system is in operation. An example is that one worker node will only be assigned to one partition.

8.5 Constraints imposed on other systems

List what constraints this system imposed on other systems.

9 Testing considerations

Explain any load testing that must be performed to evaluate the performance of this system as a whole or parts within it. Suggestions for how to test (verification and validation) this system should be included. Ways of evaluating the performance of this system should be included here.

10 Work Plan

10.1 Phase One

- Define terminology to be used to describe all the pieces of this subsystem.
- Design and deploy a database that can be loaded using the current software infrastructure. This will allow us to better understand the things that are currently being collected and also understand how the data should be

catalogued and stored for short and long term use. Code will be needed to scan current log files and to load the database. The database should include jobs run and all node assignments within the job.

- Select a distributed messaging system and establish an operating prototype. If possible, the chosen system should cover the requirements for a framework for processing the messages also (much of the coordinator role). The decision might be simpler here if a list of desired and required featured is generated.
- Establish a small cluster for development and testing. We discussed using a set of machines at Fermilab or at Vanderbilt. We have extra Infiniband cards and a switch that can be used for testing. The cards will work in PCI-X or PCI (in a degraded mode) based systems.
- Define a repository for doing development and method for building and releasing product. This might be a good time to try out subversion and SCONS.

Nearly all of these activities can be done in parallel.